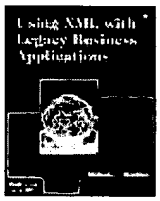


Using XML with Legacy Business Applications

By Michael C. Rawlins

Publisher	: Addison Wesley
Pub Date	: August 08, 2003
ISBN	: 0-321-15494-0
Pages	: 624
Slots	: 1.0

"This volume offers relentlessly pragmatic solutions to he applications get the most out of XML, with a breezy style going easy. Mike has lived this stuff; he has a strong com solutions and the philosophy that underlies them.">-Eve I Standards Architect, Sun Microsystems

Businesses running legacy applications that do not suppo: tough choice: Either keep their legacy applications or swi XML-enhanced applications. XML presents both challen; opportunities for organizations as they struggle with their

Does this dilemma sound familiar? What if you could ena application to support XML? You can. In *Using XML wit Applications*, e-commerce expert Michael C. Rawlins out techniques for solving day-to-day XML-related data exch Using an easy-to-understand cookbook approach, Rawlin to build XML support into legacy business applications u C++. The techniques are illustrated by building converter formats. Converting CSV files, flat files, and X12 EDI to will never be easier!

Inside you'll find:

- A concise tutorial for learning to read W3C XML s
- An introduction to using XSLT to transform betwe formats
- Simple, pragmatic advice on transporting XML do over the Internet

For developers working with either MSXML with Visual

▼ Browse By Category

Matching Categories

⊞ **Internet/Online** (1)
XML (1)
⊞ **Markup Languages** (1)
XML (1)

View All Titles
⊞ Applied Sciences
⊞ Artificial Intelligence
⊞ Business
⊞ Certification
⊞ Computer Science
⊞ Databases
⊞ Desktop Applications
⊞ Desktop Publishing
⊞ E-Business
⊞ E-Commerce
⊞ Enterprise Computing
⊞ Graphics
⊞ Hardware
⊞ Human-Computer Interaction
⊞ Internet/Online
⊞ IT Management
⊞ Markup Languages
⊞ Multimedia
⊞ Networking
⊞ Operating Systems
⊞ Programming
⊞ Security
⊞ Software Engineering

▼ Find a Specific Book

- Author
- ISBN
- Title
- Publisher

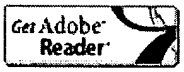
Xerces:

- See Chapter 3 for a step-by-step guide to enabling applications to export XML documents
- See Chapter 2 for a step-by-step guide to enabling applications to import XML documents
- See Chapter 5 for code examples and tips for valida documents against schemas
- See Chapter 12 for general tips on building comme application

For end users who need a simple and robust conversion u

- See Chapter 7 for converting CSV files to and from
- See Chapter 8 for converting flat files to and from
- See Chapter 9 for converting X12 EDI to and from
- See Chapter 11 for tips on how to use these techniq complex format conversions

URL http://proquest.safaribooksonline.com/0321154940



75 Arlington Street, Floor 3
Boston, MA 02116
1-800-889-3358

Copyright

Library of Congress Cataloging-in-Publication Data

Rawlins, Michael C.

Using XML with legacy business applications / Michael C. Rawlins.

p. cm.

ISBN 0-321-15494-0 (Paperback : alk. paper)

1. XML (Document markup language) 2. Business—Computer programs. I. Title.

QA76.76.H94R375 2003

005.7'2—dc21

2003010094

Text printed on recycled paper

1 2 3 4 5 6 7 8 9 10—CRS—0706050403

First printing, August 2003

Dedication

This book is dedicated to all of the small to medium enterprises and others who are still waiting to see the benefits of XML

and

to Dick, Don't Panic!

XML to Flat File: Detail Design

Main Program

The shell main program functions are very similar to those of the XML to CSV converter.

Logic for the Shell Main Routine for XML to Flat

```
Arguments:
  Input Directory Name
  Output Flat File Name
  File Description Document Name

Options:
  Validate input
  Help

Validate and process command line arguments
IF help option specified
  Display help message
  Exit
ENDIF
Open output file
Create new FlatTargetConverter object, passing:
    Output Stream
    File Description Document Name
Set up implementation dependent DOM environment for loading,
    parsing, and validating input documents
Open input directory
Get first file from input directory
DO for all files in input directory
  Input Document <- Load input file, validating it if requested
  Call FlatTargetConverter processDocument method, passing
      the Input Document
  Increment number of documents processed
ENDDO
Close output file
Display completion message with number of documents processed
```

FlatTargetConverter Class (Extends TargetConverter)

Overview

The FlatTargetConverter again is very similar to the CSVTargetConverter. However, like the FlatSourceConverter, we use a recursive algorithm to process the logical record groups in input documents. We use the base TargetConverter's processGroup method, described in this subsection.

Attributes:

- None

Methods:

- Constructor
- processDocument
- processGroup (base class method)

Methods

Constructor

The constructor method for our FlatTargetConverter object sets up that object and the FlatRecordWriter object.

Logic for the FlatTargetConverter Constructor Method

```
Arguments:
  Output Stream Flat Output file
  File Description Document Name

Call base class constructor, passing File
    Description Document Name
Create FlatRecordWriter object, passing:
    File Description DOM Document and Output Stream
```

processDocument

The bulk of the processing is performed in the FlatTargetConverter's processDocument and processGroup methods. This method converts one input XML Document and writes it to the flat file output stream based on the input parameters. Most of the actual work is done in the processGroup method.

Logic for the FlatTargetConverter processDocument Method

```
Arguments:
  DOM Document Input Document

Returns:
  Error status or throws exception

Root Element <- Get Document's documentElement attribute
Root Element Name <- Get Root Element's tagName attribute
IF Root Element Name != Grammar Root Element Name
  Return error
ENDIF
Call processGroup, passing Root Element and Grammar Element
Return success
```

processGroup (Base Class TargetConverter Method)

As with the processGroup method of the SourceConverter we use a recursive algorithm, but the details are a little bit different for the termination case. This method is used for both flat file and EDI conversions.

Logic for the FlatTargetConverter processGroup Method

```
Arguments:
  DOM Element Parent Element
  DOM Element Group Grammar

Returns:
  Error status or throws exception

Record Grammar Node <- Get firstChild from Grammar Element
    skipping over non-Element Nodes
Record Grammar Element <- Record Grammar Node

// Process the Group's starting Record Element
Record Element <- Get first childNode from Parent Element,
    skipping over non-Element Nodes

Call RecordWriter's parseRecord, passing Record Element and
    Record Grammar Element
Call RecordWriter derived class's writeRecord, passing Record
    Grammar Element

// This advance makes sure that we don't repeat starting record
// in group
Record Grammar Element <- Get next Record Element from
    Group Grammar, skipping over non-Element Nodes
Grammar Tag = call Record Grammar Element's getAttribute for
    "ElementName"

Record Element <- Get Record Element's nextSibling, skipping over
    non-Element Nodes

DO until all child Elements of Parent have been processed
  Record Tag <- call Record Element's getNodeName
  DO until Grammar Tag = Record Tag
    Record Grammar Element <- Get next Record Element from
        Group Grammar, skipping over non-Element Nodes
    IF Record Grammar Element is NULL
      Return error  // This record is not part of the group
    ENDIF
  ENDDO
  Grammar Element Name <- Call Record Grammar Element's
      getNodeName
  IF Grammar Element Name = "GroupDescription"
    Call processGroup, passing Record Element and Record Grammar
        Element
  ELSE
    Call RecordWriter's parseRecord, passing Record Element and
        Record Grammar Element
    Call RecordWriter derived class's writeRecord, passing
        Record Grammar Element
  ENDIF
  Record Element <- Get Record Element's nextSibling, skipping
      over non-Element Nodes
ENDDO
```

This method is called the first time from processDocument, which passes it the root Element of the Document and the complete Grammar Element. It processes the Element that represents the header record of the logical document, then advances to process its sibling records and groups. Record Elements are processed in the same fashion as the header record Element. Again, recall that in our XML representation of flat files a group of records is explicitly represented by an Element, with the group members (records or other groups) as child Elements. When we encounter an Element that represents a group (that is, when the Element's grammar Element has the name "GroupDescription"), we make a recursive call.

The termination cases are a bit different than those in the FlatSource Converter. In the XML representation the groups are explicitly represented, so we don't have a normal case of encountering an Element that isn't in the current group grammar. If we do hit one, this is an error; we return an error or throw an exception. For normal processing the method continues execution until all the Elements in the source document's group have been processed, then we return back to the caller.

FlatRecordWriter Class (Extends RecordWriter)

Overview

As is the case with the CSVRecordWriter class, the FlatRecordWriter is derived from the RecordWriter base class (see Chapter 6). The most important method developed in this class is the writeRecord method. We use the parseRecord method inherited from the RecordWriter base class.

Attributes:

- Integer Fixed Record Length
- Integer Record ID Field Offset
- Integer Record ID Field Length

Methods:

- Constructor
- writeRecord

Methods

Constructor

The logic for the FlatRecordWriter constructor method is essentially the same as that for the FlatRecordReader.

Logic for the FlatRecordWriter Constructor Method

```
Arguments:
  DOM Document File Description Document
  Output Stream

Call RecordWriter base class constructor, passing File
    Description Document and Output Stream
Record Format Element <- Get "RecordFormat" Element from File
    Description Document
Child Element <- Get first childNode from Record Format Element,
    advancing over non-Element Nodes
Fixed Record Length <- 0
IF Child Element NodeName = Fixed
  Fixed Record Length <- Call Child Element's getAttribute for
      "Length"
ELSE
  Record Terminator <- Call Child Element's getAttribute for
      "RecordTerminator"
  Call setTerminator to set the Record Terminators
ENDIF
Tag Info Element <- Get "RecordTagInfo" Element from File
    Description Document
Record ID Field Offset <- Call Tag Info's getAttribute for
```

```
   "Offset"
Record ID Field Length <- Call Tag Info's getAttribute for
   "Length"
```

writeRecord

The writeRecord method handles the aspects of record formatting unique to each legacy format. We should again note two restrictions mentioned at the beginning of the chapter. The record identifier field will be converted if it is present in the input XML document, but the value will be overridden by the value specified in the grammar. If a fixed record length is specified the record is initialized to ASCII spaces. Variable length records are initialized with the null character.

The approach for writing a record to flat files is somewhat different than the approach we used for writing to CSV files. For CSV files we walked through the DataCell Array and built the output buffer from its contents. Column number was key, and if a column was missing, a column delimiter was inserted. Due to the requirement to fill missing fields with a default fill character, we must take a different approach and drive the conversion from the record grammar instead of the contents of the DataCell Array. We will cycle though the FieldDescription Elements of the RecordDescription, find matching fields in the DataCell Array, and build the output buffer accordingly. If a field in the grammar is not present in the DataCell Array built from the input, we fill it using the fill character specified in the grammar.

Logic for the FlatRecordWriter writeRecord Method

```
Arguments:
  DOM Element Record Grammar

Returns:
  Error status or throws exception

IF Fixed Record Length > 0
  Initialize Output Record Buffer to null characters
ELSE
  Initialize Output Record Buffer to ASCII spaces
ENDIF
Record Length <- 0
Cell Array Index <- 0
Field Grammar NodeList <- Call Record Grammar's
    GetElementsByTagName for "FieldDescription"
DO for all items in Field Grammar NodeList
  Field Grammar Element <- next item from Field Grammar NodeList
  Grammar Field Number <- call Field Grammar Element's
     getAttribute for "FieldNumber"
  Offset <- Call Grammar Element's getAttribute for "Offset"
  Length <- Call Grammar Element's getAttribute for "Length"
  Field Written <- false
  IF (Index <= Highest Cell)
    Field Number <- call CellArray[Index] getFieldNumber
    IF (Field Number = Grammar Field Number)
      Call CellArray[Index] fromXML
      Call CellArray[Index] prepareOutput
      Field Contents <- Call CellArray[Index] getField
      Output Buffer <- Insert Field Contents at Offset for Length
      Clear CellArray[Index]
      Increment Cell Array Index
      Field Written <- true
    ENDIF
  ENDIF
  IF ( Field Written = false)
    Fill Character <- Call Grammar Element's getAttribute for
       "FillCharacter"
    Output Buffer <- Insert Fill Character at Offset for Length
  ENDIF
  Working Length <- Offset + Length
  IF (Working Length > Record Length)
    Record Length = Working Length
  ENDIF
ENDDO
```

```
Highest Cell <- -1
Record ID Value <- Call Record Grammar Element's getAttribute
   for "TagValue"
Output Buffer <- Insert Record ID Value at Record ID Field
   Offset for Record ID Field Length
IF Fixed Record Length = 0
  Append base RecordHandler's Record Terminators to Output
     Buffer
  Increment Record Buffer Length
ENDIF
Call language's write routines to do physical write of
   Output Buffer for Record Length
Return success
```

URL http://proquest.safaribooksonline.com/0321154940/ch08lev1sec7